Creative Programming for Young Minds

. . . on the TI-99/4A ™

Volume VII

CREATIVE
PROGRAMMING INCORPORATED
A SUBSIDIARY OF R.V. WEATHERFORD CO.

CREATIVE Programming for Young Minds

CREATIVE Programming for Young Minds didn't just happen. It represents the harvested fruit of an idea planted several years ago by Dr. Henry A. Taitt. He saw the pressing need for an enrichment program for young children that would help prepare them for the future they would be instrumental in shaping.

It was cultivated by Marilyn Buxton, whose deep interest in early childhood learning enabled her to find ways to teach primary children to program microcomputers.

It was fertilized by Devin Brown, with his lively wit and creative writing style. He gave it the nutrients it needed to appear in printed form to be shared. His shadow is cast over most of the later authors who patterned their style and examples after his original writings.

It was cared for by Howard Smith, Charles Miller, George Kolopanis, Alverta Darding, Lea Ann Hummel, Robin Koch and others, who worked with it in the lab helping to remove the bugs that would stunt its growth.

It was harvested by Nancy Taitt, Marilyn Hoots, Wayne Owens, Diane ZuHone, and others who typed and phoned and talked with people to spread the word and create a market for the final fruit.

And most important of all were the CHILDREN who tried and tested the materials that were produced. They shared their likes and dislikes, and made certain that everything that was included could be done by young minds.

These books were not created by a publisher to be sold to schools, where they would be used on children. They were instead, created from the successes of children, edited by the concerns of parents, and then offered to anyone that wishes to enrich the minds of young children.

If you elect to use these materials, then you assume the responsibility to encourage independent thought, reward creativity, enhance reasoning and logic, and above all, be forever open to alternate ways to solve problems.

If you do this, your own rewards will be found in the faces of the children you serve.

CREATIVE
PROGRAMMING INCORPORATED
A SUBSIDIARY OF R.V. WEATHERFORD CO.

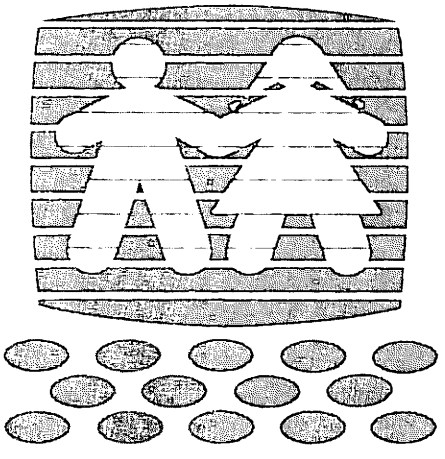
(217) 348-1451

Creative Programming for Young Minds

. . . on the TI-99/4A ™

Volume VII

by Leonard Storm



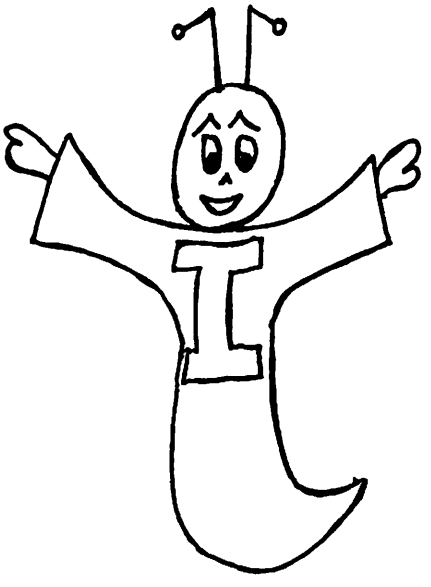
CREATIVE PROGRAMMING FOR YOUNG MINDS

...ON THE TI-99/4A

VOLUME VII

T A B L E O F C O N T E N T S

LESSON #24: SPRITES



CONGRATULATIONS, TI-99/4A MASTER PROGRAMMER! YOU HAVE COMPLETED VOLUME VI AND ARE READY FOR THE SPRITELY ADVENTURES AHEAD. IN THIS VOLUME, WE WILL EXPLORE SOME CAPABILITIES OF THE TI EXTENDED BASIC COMMAND MODULE. TO BEGIN, INSERT THE EXTENDED BASIC MODULE INTO THE COMPUTER AND THEN TURN ON YOUR COMPUTER.

In this lesson, you will learn how to create smoothly moving graphics characters called sprites. These sprites are defined using the CALL SPRITE subprogram. The format of the CALL SPRITE subprogram is shown below:

CALL SPRITE (#*number*,*character*,*color*,*dotrow*,*dotcol*,*rowvel*,*colvel*)

The first variable, *number*, is a numeric expression from 1 to 28. That is, up to 28 sprites can be defined at any one time. The # sign must be present in front of the numeric expression.

The second variable, *character*, is the character code which defines the shape of the sprite. This number may be any integer from 32 to 143.

The third variable, *color*, defines the foreground color of the sprite. The background color of a sprite is always transparent. COLOR may be any numeric expression from 1 to 16.

The next two variables determine the starting position of the sprite. *Dotrow* is a number from 1 to 256 (but 193 through 256 are off the bottom of the screen) and *dotcol* is a number from 1 to 256. The DOTCOL=1,DOTROW=1 position is the upper left-hand corner of the screen. DOTROW and DOTCOL actually specify the position of the upper left-hand corner of the character or characters which define the sprite.

The last two variables of the CALL SPRITE subprogram are optional. If present, they specify the row and column velocity of the sprite. If these variables are not present, then the sprite will be stationary. *Rowvel* and *colvel* may be any number from -128 to 127.

Now type the following program into the computer and RUN it:

```
10 CALL CLEAR
20 INPUT "CHARACTER ":Z
30 CALL SPRITE (#1,Z,16,80,80,10,10)
40 GOTO 10
```

Input the following numbers and record the sprite shape.

CHARACTER	SHAPE
65	_____
34	_____
98	_____
72	_____
_____	_____
_____	_____

Notice that once the sprite is defined in statement 30, it continues to move with the specified velocity until the sprite parameters are changed.

Also, notice that every time you run the CALL SPRITE subprogram, the new character starts again at dot row 80 and dot column 80.

Next, change statement 30 to:

```
30 CALL SPRITE(#1,66,16,80,80,X,Y)
```

and change statement 20 to:

```
20 INPUT "XVEL,YVEL":X,Y
```

RUN the program.
On the next page, INPUT the values listed for XVEL and YVEL.

XVEL	YVEL	DIRECTION OF MOTION (Draw an arrow.)
0	0	
10	0	
-10	0	
0	10	
0	-10	
-30	-30	

Now change statement 20 to:

```
20 INPUT "SPRITE NUMBER":S
```

and change statement 30 to:

```
30 CALL SPRITE(#S,66,16,80,80,10,10)
```

Next, RUN the program. Input the following numbers (one at a time):

S = 1, 1, 1, 2, 3, 28, 15, 4, 5, 9, 8, 7

Note that entering the same number several times redefines the same character and starts its motion at the position DOTROW=80 and DOTCOL=80.

After you enter the numbers above, you should have 10 sprites moving diagonally across the screen.

What happens if you input a number outside the range from 1 to 28? _____

_____

Now change statement 30 to:

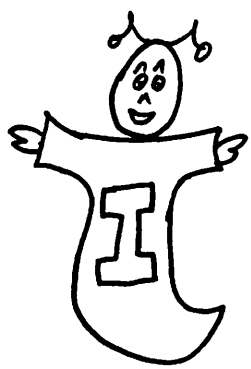
```
30 CALL SPRITE(#S,66+S,16,80,80,0,10)
```

RUN the program and input the following numbers:

S = 1, 3, 9, 21

When you can see all the sprites on the screen at one time, enter another sprite, say S = 11.

Notice that when you entered the last sprite, #11, one of the other sprites disappears. The sprite that has disappeared still exists, it's just invisible. The rule is:



ONLY 4 SPRITES WILL BE VISIBLE ON ANY ONE ROW OF THE SCREEN. THE LOWEST NUMBERED 4 SPRITES. WILL BE VISIBLE. HIGHER NUMBERED SPRITES WILL BE INVISIBLE.

Now add the following program statements:

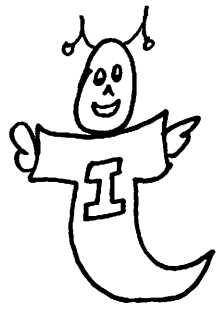
```
25 IF S=0 THEN 45
45 INPUT "DELETE SPRITE":M
50 CALL DELSPRITE(#M)
55 GOTO 10
```

RUN the program again and input:

S = 21, 3, 9, 1.

Then input S = 11 and observe that sprite #21, the W disappears.

Next, input S = 0 and when you get the DELETE SPRITE message, input a 1. This causes sprite #1 to be undefined. It disappears because it no longer exists. Now, the W reappears since only four sprites exist on that row.



THE CALL DELSPRITE COMMAND IS USED TO DELETE CURRENTLY DEFINED SPRITES.

Now change the program again. Change statement 30 to:

```
30 CALL SPRITE (#S,66+S,16,80,80,-10,0)
```

Input as many sprites numbers as you wish. Notice that none of the sprites disappear. All 28 sprites can exist and be visible in any one column.

Now add the following program statement:

```
1 CALL MAGNIFY(2)
```

RUN the program again and input any numbers that you wish. Notice that the CALL MAGNIFY(2) statement causes all the sprites to have twice the size that they had before.

Finally, change statement 1 to:

```
1 CALL MAGNIFY(1)
```

RUN the program again and notice the size of the sprites.

Instead of using the standard character set, let's define a sprite of our own design. It's really quite easy to do. We just use the CALL CHAR subprogram

A program example is shown below. Type it into the computer and then RUN it.

```
10 CALL CLEAR
20 CALL CHAR(96,"3C7EE7DBDBE77E3C")
30 CALL SPRITE(#1,96,7,60,80,16,15)
40 GOTO 40
```

Now add the following program lines:

```
40 CALL CHAR(40,"FFFFFFFFFFFFFFFF")
50 CALL COLOR(2,11,11)
60 CALL HCHAR(10,1,40,32)
70 GOTO 70
```

RUN the program.

Here's how the program works:

Statement 20 defines the sprite character shape.

Statement 30 defines the sprite characteristics. The sprite is to be red, start at DOTROW 60 and DOTCOL 80, and move with speed 16 downward and 15 to the right.

Statement 40 defines character 40 to be a solid square.

Statement 50 causes set 2 (which contains character 40) to be colored dark yellow.

Statement 60 prints 32 of the yellow squares in a horizontal line beginning at row 10 and column 1.

Notice that the sprite that was defined appears to move over the dark yellow band.

What change would you have to make to cause the sprite to have twice the size it has now? Include a line in your program which will make the sprite twice as large.

Write your program line: _____

Now change statement 30 to:

```
30 CALL SPRITE (#1,96,7,60,80,16,15,#2,96,7,62,100,16,15)
```

RUN the program again.

Notice that statement 30 defines two sprites at once.

See if you can alter statement 30 so that sprite #1 is black and sprite #2 is red. Show the revised statement 30 on the line below. Then RUN the program to check it out.

_____

Now let's write a bouncing ball program. Later we will add a paddle and make a game out of it.

Type the following program code into your computer:

```
10 CALL CLEAR
20 A$="3C7EFFFFFFFF7E3C"
30 CALL COLOR(2,5,8)
40 CALL SCREEN(8)
50 B$="FFFFFFFFFFFFFFFF"
60 CALL CHAR(40,A$)
70 CALL COLOR(3,2,8)
80 CALL CHAR(48,B$)
```

Statement 20, 30, and 60 define the ball character. Statement 50, 70, and 80 define the character which will be used as a border.

Also type in the following lines:

```
90 CALL HCHAR(1,1,48,64)
100 CALL HCHAR(23,1,48,64)
110 CALL VCHAR(1,1,48,48)
120 CALL VCHAR(1,31,48,48)
130 CALL MAGNIFY(2)
140 CALL SPRITE(#5,40,7,80,80,8,10)
```

Statements 90 through 140 print a solid border around the screen.

Statement 130 causes sprites to be double sized and statement 140 prints the ball sprite on the screen. The sprite is started at position DOTROW = 80 and DOTCOL = 80. Its speed is 8 units downward and 10 units to the right.

Now type in the following code:

```
150 AX=8::AY=10::VX=8::VY=10
160 CALL POSITION(#5,X,Y)
170 IF X>160 THEN VX=-AX ELSE IF X<20 THEN VX=AX
180 IF Y>220 THEN VY=-AY ELSE IF Y<20 THEN VY=AY
190 CALL MOTION(#5,VX,VY)::GOTO 160
```

Statement 150 sets VX equal to the vertical speed of the sprite and VY equal to the horizontal speed. The double colon (::) is a statement separator which allows more than one statement to be placed in a BASIC line.

Statement 160 asks for the position of sprite #5. This statement sets X equal to the vertical position of the sprite and Y equal to the horizontal position of the sprite. (The position of a sprite is its upper left-hand corner.)

Statement 170 then checks the value of X against the horizontal borders of the screen. If X > 160, then the sprite has hit the screen border on the bottom. If X < 20, then the sprite has run into the top border. For either of these conditions, the vertical direction should be reversed. That is, set VX= -AX or + AX.

NOTE: TI EXTENDED BASIC ALSO ALLOWS LOGICAL EXPRESSIONS TO BE CONNECTED WITH LOGICAL OPERATORS (AND, OR, NOT, XOR).

Statement 180 checks to see if the ball sprite has run into the side borders. If it has, then the horizontal motion of the sprite should be changed: VY= -AY or + AY.

Statement 190 contains another special subprogram used with sprites. The CALL MOTION subprogram allows the velocity of the sprite to be changed. After statement 190 is executed, the velocity of sprite #5 will be VX in the vertical direction and VY in the horizontal direction.

Next statement 190 causes a jump back to line 160 where the position of sprite #5 is again determined.

Statements 160 through 190 form a repeating loop.

RUN the program and observe the smooth bouncing action of the ball sprite.

Now stop the program.

Let's define a paddle-like sprite to control the bouncing of the ball.

Type the following program lines into the computer:

```
25 P$="FFFFFF0000FFFFFF"
60 CALL CHAR(40,A$,44,P$)
140 CALL SPRITE(#5,40,7,80,80,8,10,#4,44,15,140,120)
```

Statement 25 defines the paddle shape while statement 60 assigns this string to the character code 44. Note that more than one character code assignment can be made in the same CALL CHAR subprogram.

Statement 140 has been expanded to define both sprite #5 and sprite #4. Sprite #4 is the paddle character, 44, and is defined to be gray (15), and is located at location 140 down and 120 to the right.

RUN the program again and observe the effects of the changes which have been made.

To get the paddle to move, modify the following program lines as shown:

```
160 CALL POSITION(#5,X,Y)::CALL KEY(1,K,S)::IF S=0
THEN PV=0 ELSE PV=20*(K-2.5)
190 CALL MOTION(#5,VX,VY,#4,0,PV)::GOTO 160
```

Statement 160 determines the position of the ball. Then the CALL KEY subprogram sets K equal to the key code of the key pressed on the left-hand side of the keyboard (unit 1). S is set equal to the status of the keyboard. If S=0, then no key is currently being pressed. The IF-THEN portion of statement 160 will then set PV=0 (paddle velocity=0). If S does not equal zero, then PV will be set equal to 20*(K-2.5).

Statement 190 sets the velocity of the two sprites.

Now RUN the program again. Use the two arrow keys (← →) to control the velocity of the paddle. (K=2 for ←; K=3 for →).

NOTE: PRESSING ANY KEY OTHER THAN THE TWO ARROW KEYS MAY CAUSE THE PROGRAM TO "CRASH".

We now need to add the statement which will cause the ball to bounce off of the paddle.

Type in the following line:

```
185 CALL COINC(#4,#5,T,C)::IF C=-1 THEN VX=-VX::CALL
SOUND(-50,880,0)::T=0 ELSE T=8
```

RUN the program again. Use the arrow keys to move the paddle. Try to hit the ball with the paddle.

Statement 185 contains a COINCidence subprogram which checks to see whether sprite #4 and sprite #5 are at the same screen position.

The third variable, T, is a tolerance variable. If sprite #4 and sprite #5 are within T screen dots, then the sprites are considered to be coincident. The CALL COINC subprogram sets the numeric variable C equal to -1 if the sprites are coincident; otherwise, it sets C equal to 0.

The next part of statement 185 checks the value of C. If C = -1, then the sprites were coincident. The program then causes the vertical motion to be reversed, a sound to be played, and the tolerance variable to be set to zero. If C doesn't equal -1, then T is set to 8.

EXERCISE 24-1

Now let's go back and review the new commands that you have learned. They are listed below for your convenience:

CALL SPRITE	CALL MOTION
CALL DELSPRITE	CALL COINC
CALL MAGNIFY	CALL POSITION

Write a program to do the following:

Create an arrow shaped character (⟶) using the CHAR subprogram. Then use the sprite subprogram to create an arrow shaped sprite having a red color. The sprite should be located at DOTROW=100 and DOTCOL=20. The velocity of the arrow sprite should be zero.

Use a statement like 1000 GOTO 1000 at the end of your program.

RUN the program to see that it works, then write your program below:

_____

_____

_____

_____

_____

Now add a statement to your program to cause the sprite to move to the right with a velocity of 35. Show the program line in the space below when you have it working properly. _____

EXERCISE 24-1 (CONT.)

What statement could be added to your program to make the sprite be double sized? Add the statement to your program. Write it on the line below.

_____

Now at the end of your program, but before the last infinite loop, include the delay loop shown below:

```
FOR DELAY=1 TO 1000::NEXT DELAY
```

After the delay loop, include a CALL MOTION statement to stop the motion of the sprite. Then add another delay loop. Finally, use the DELSPRITE subprogram to delete the sprite. Show the additional program lines below. (The 1000 GOTO 1000 statement should still be the last program line.)

_____

_____

_____

_____

_____

_____

_____

EXERCISE 24-2

In this exercise, you will investigate the COINC, POSITION, and LOCATE subprograms. To do this, type in the following program lines:

```
5 CALL CLEAR
10 CALL CHAR(40,"CCCC3333CCCC3333")
20 CALL SPRITE(#1,40,2,80,100,#2,40,16,80,100)
30 CALL MAGNIFY(2)
40 INPUT "X,Y":X,Y
50 CALL LOCATE(#1,X,Y)
60 CALL COINC(#1,#2,10,I)
70 CALL POSITION(#1,A,B)
80 PRINT "POSITION ";A;B
90 PRINT "COINC ";I
100 FOR J=1 TO 1000
110 NEXT J
120 CALL CLEAR
130 GOTO 40
```

A description of the program follows:

Statements 10 through 30 define the sprite characteristics.

Statement 40 allows you to input new position coordinates for sprite #1.

Statement 50 actually repositions the sprite. The first variable of the CALL LOCATE subprogram specifies

EXERCISE 24-2 (CONT.)

the sprite that will be positioned. The next 2 variables specify the dot row and dot column of the upper left-hand corner of the sprite.

Statement 60 looks to see whether the upper left-hand corners of the sprites are within 10 dots of each other. If they are, then I will be set to -1, otherwise, it will be set to 0.

Statement 70 determines the position of sprite #1.

Statements 80 and 90 print out the position and coincidence information generated in 60 and 70.

Statements 100 and 110 cause a delay before statement 120 clears the screen of print and statement 130 loops back to the input statement.

RUN the program and input position data for sprite #1. Record the value generated by the COINC subprogram.

X	Y	COINC

EXERCISE 24-2 (CONT.)

Now rewrite the last program using :: to condense the program to as few program lines as possible. RUN the program to make sure that it works. Then write the program below.

EXERCISE 24-3

In this exercise, you will learn about the CALL PATTERN subprogram which allows you to quickly change the shape of a previously defined sprite.

The form of the PATTERN subprogram is shown below:

CALL PATTERN(#*sprite*,*character*,*value*)

The first variable specifies the number of the sprite whose character code is to be changed. The second variable is the new character code for the sprite.

The CALL PATTERN subprogram is illustrated in the following program. Type it into the computer and RUN it.

```
5 CALL CLEAR
10 CALL CHAR(40,"101010FE10101000",42,"8244281028448200")
::CALL MAGNIFY(2)
20 CALL SPRITE(#1,40,7,10,10,10,10)
30 P=42
35 FOR T=1 TO 500::NEXT T
40 CALL PATTERN(#1,P)
50 P=40
55 FOR T=1 TO 500::NEXT T
60 CALL PATTERN(#1,P)
70 GOTO 30
```

Statement 10 defines two different characters.
Statement 20 defines a sprite using one of these characters.
The rest of the program alternately assigns one or the other shape to the moving sprite.

EXERCISE 24-3 (CONT.)

Now it's your turn. Write a program using the CALL PATTERN subprogram which defines a moving block that grows and shrinks as it moves along. Use at least four different codes to define different sized blocks.

When your program works properly, record it on the lines below.

EXERCISE 24-4

In this exercise, you will learn about another sprite subprogram: CALL DISTANCE. This subprogram has the form shown below:

CALL DISTANCE(#*sprite*,#*sprite*,*numeric variable*)

OR

CALL DISTANCE(#*sprite*,*dotrow*,*dotcol*,*numeric variable*)

The first form of the subprogram finds the distance squared between the two sprites listed as variables of the subprogram. NUMERIC VARIABLE is set equal to the calculated distance squared.

The second form of the subprogram finds the distance squared between the sprite specified and the screen position specified. NUMERIC VARIABLE is set equal to the distance squared.

Type the following program into the computer. It illustrates the DISTANCE subprogram.

```
10 CALL CLEAR
20 CALL CHAR(40,"C0C0000000000000")
30 CALL SPIRTE(#1,40,2,40,160,#2,40,16,40,160)
40 INPUT "R1,C1 ":R1,C1
50 INPUT "R2,C2 ":R2,C2
60 CALL SPRITE(#1,40,2,R1,C1,#2,40,16,R2,C2)
70 CALL DISTANCE(#1,#2,D)
```

Keep going. ——————→

EXERCISE 24-4 (CONT.)

```
80 PRINT "DISTANCE SQUARED ":D
90 FOR I=1 TO 1000
100 NEXT I
110 GOTO 10
```

RUN the program. Input the following data and fill in the value for distance squared.

R1	C1	R2	C2	D
40	160	40	160	
40	160	40	161	
40	160	40	162	
40	160	40	165	
40	160	41	160	
40	160	42	160	
40	160	50	160	
40	160	30	160	
40	160	41	161	
40	160	42	162	

EXERCISE 24-5

In this exercise, you will learn how to make GIANT-SIZED sprites. It's really quite easy to do.

Thus far, you have learned how to create sprites consisting of a single character. To create larger sprites, one must define four characters.

For example, one could use the character codes 40, 41, 42 and 43 to define a large sprite. (Note that the numbers should be consecutive and the first one should be divisible by 4.)

The first code number defines the upper left-hand corner of the sprite. The second character code defines the lower left-hand corner of the sprite. The third number corresponds to the lower right-hand corner. This is illustrated below:

40	42
41	43

The following program illustrates how one may create a 4-character sprite:

```
10 CALL CLEAR
20 CALL CHAR(44,"FF00FF00FF00FF00")
30 CALL CHAR(45,"AAAAAAAAAAAAAAAA")
40 CALL CHAR(46,"FFFFFFFFFFFFFFFF")
```

Keep going. ——————→

EXERCISE 24-5 (CONT.)

```
50 CALL CHAR(47,"00003C3C3C3C0000")
60 CALL SPRITE(#1,44,7,40,40,10,10)
70 FOR J=1 TO 4
80 CALL MAGNIFY(J)
90 FOR I=1 TO 1000::NEXT I
100 NEXT J
110 GOTO 70
```

RUN the program. Observe the various sprite sizes.

When J=1, statement 80 causes the sprite to consist of one character and be single-sized.

When J=2, the sprite will consist of one character and be double-sized.

MAGNIFY(3) causes the sprite to consist of 4 characters, each single-sized.

MAGNIFY(4) causes the sprite to consist of 4 characters, each double-sized.

Note statements 20 through 50 could be replaced by a single CALL CHAR statement. The additional forms are shown below.

```
CALL CHAR(44,"FF00FF00FF00FF00",45,"AAAAAAAAAAAAAAAA",
46,"FFFFFFFFFFFFFFFF",47,"00003C3C3C3C0000")
```

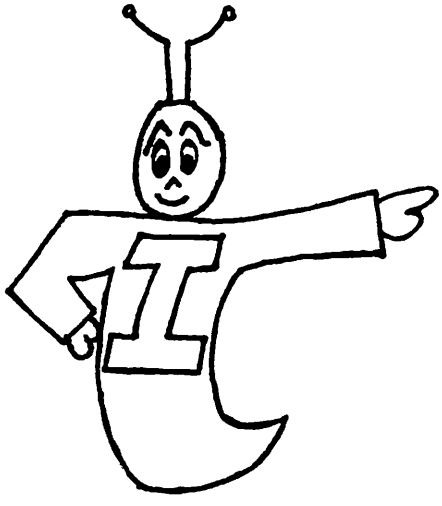
OR

```
CALL CHAR(44,"FF00FF00FF00FF00AAAAAAAAAAAAAAAAFFFFFFFF
FFFFFFFF00003C3C3C3C0000")
```

EXERCISE 24-5 (CONT.)

Now it's your turn. Write a program which creates a GIANT-SIZED sprite. Use the CALL PATTERN and CALL MOTION subprograms to introduce linear motion and the effect of rotation. Write the program below.

LESSON #25: DISPLAY



IN THIS LESSON, YOU WILL FIND OUT ABOUT THE VERSATILE DISPLAY AT COMMAND WHICH ALLOWS TEXT TO BE PLACED AT ANY POSITION ON THE SCREEN WITHOUT CAUSING THE SCROLLING ASSOCIATED WITH THE PRINT COMMAND.

A simple form of the DISPLAY command is illustrated in the program below. Type it into the computer and RUN it.

```
10 CALL CLEAR
20 A$="HELLO, MY NAME IS TEX."
30 B$="SEE, I CAN PRINT ANYWHERE!"
40 DISPLAY AT(6,5):A$
50 DISPLAY AT(23,2):B$
60 DISPLAY AT(15,10):B$
70 GOTO 70
```

The numbers included inside the parenthesis tell where to begin printing the desired message. For example,

```
DISPLAY AT(1,2):"HELLO";X$
```

would print the message HELLO at row 1, column 2. The string, X$, would be printed in the next column to the right of the O in HELLO since a semi-colon is used as the separator.

Now, add a line to the program that will cause the computer to print a message at row 17, column 8. Show your working program line here: _____

Now change statement 40 to the following:

```
40 DISPLAY AT(6,5)BEEP:A$
```

RUN the program again.

BEEP causes a short tone to be played when the data is displayed.

Next, add the following program lines:

```
55 FOR I=1 TO 1000::NEXT I
60 DISPLAY AT(15,10)ERASE ALL:B$
```

RUN the program again and notice the effect of the ERASE ALL.

Now type in the following program:

```
10 CALL CLEAR
20 A$="125"::B$="7649"
30 DISPLAY AT(12,14)BEEP:B$
40 DISPLAY AT(12,11)BEEP:A$
50 GOTO 50
```

RUN the program. Notice that the information printed in statement 30 gets erased by statement 40.

To inhibit this erasure, change statement 40 to:

```
40 DISPLAY AT(12,11)SIZE(5)BEEP:A$
```

RUN the program again.

Notice that statement 40 now only erases 5 spaces before printing the A$ data.

NOTE: IF THE SIZE OPTION IS NOT PRESENT, THE REST OF THE ROW WILL BE ERASED BEGINNING AT THE COLUMN SPECIFIED IN THE DISPLAY COMMAND.

Now change statements 30 and 40 to:

```
30 DISPLAY AT(24,4)BEEP:B$
40 DISPLAY SIZE(3)BEEP:A$
```

RUN the program again. Where is the information printed this time? _____

Now add this line and RUN the program again.

```
45 DISPLAY AT(24,1):A$::GOTO 45
```

Notice that using DISPLAY without an AT causes the line to be printed in the 24th row, first column and also causes an upward scroll to the 23rd row.
Statement 45 causes no such scroll.

Now for something different. . .

You have no doubt noticed that computer calculations often give results containing many decimal digits and yet sometimes it is desirable to ignore most of those digits.

For example, if A=124.1234567, you may desire to print A as 124.12 and ignore the rest. It is possible to do this

using a special format option of TI EXTENDED BASIC. The following program shows how this is done. Type it into the computer and RUN it.

```
10 CALL CLEAR::INPUT A
20 DISPLAY AT(5,5):USING "##.###":A
30 FOR I=1 TO 1000::NEXT I
40 GOTO 10
```

Input the following data and record the computer's response.

A	RESPONSE
1	
12	
12.1	
12.12	
64.123	
82.1234	
10.0000	
100	
100.11	
-1	
-12	
.9995	
99.9994	
99.9995	

Now change statement 20 so that the computer BEEPs when it displays the data. Show the working program line below.

Instead of putting the number format in the USING clause, one may put it in an IMAGE statement. The variable of the USING clause is then the line number of the IMAGE statement. This option is illustrated in the program which follows. Type it into the computer and RUN it.

```
1 IMAGE ##.#,##.##,##.###
10 CALL CLEAR::INPUT A,B,C
20 DISPLAY AT(1,1):USING 1: A,B,C
30 FOR I=1 TO 1000::NEXT I
40 GOTO 10
```

INPUT	RESPONSE
10,11,12	
1,2,3	
123,2,4	

Statement 20 displays the message beginning at row 1, column 1, using the format specified in statement 1.

Notice that the # symbol gets replaced by the numerical value of the variable to be displayed. The periods, commas, etc. in the IMAGE statement are printed just as they are found in the IMAGE statement.

Now make the following changes:

```
1 IMAGE HELLO, ##########
10 CALL CLEAR::INPUT A$
20 DISPLAY AT(1,1):USING 1:A$
```

RUN the program again. Input TEX at the question mark.

Now change statement 20 to:

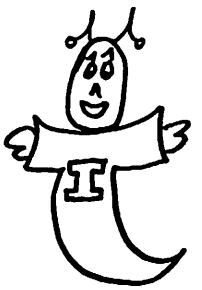
```
20 PRINT USING 1:A$
```

RUN the program again. Use TEX as the input.

Next, type in and RUN the following program:

```
1 IMAGE ##.########
10 CALL CLEAR::INPUT A
20 PRINT USING 1:A
30 FOR I=1 TO 1000::NEXT I
40 GOTO 10
```

A	RESPONSE
1	_____
123	_____
123.123E100	_____
-1E-5	_____
_____	_____
_____	_____



TI EXTENDED BASIC ALSO ALLOWS ONE TO INPUT DATA FROM ANYWHERE ON THE SCREEN. THE COMMAND WHICH ALLOWS THIS IS CALLED ACCEPT AT.

The ACCEPT command is illustrated in the program below. Type it into the computer and RUN it.

```
10 CALL CLEAR
20 ACCEPT AT(2,20):M
30 DISPLAY AT(20,2)BEEP:M
40 GOTO 20
```

Input the following data and write the computer's response.

DATA	RESPONSE
1	_____
23	_____
1234567890	_____
112233445566	_____
-1.23E4	_____
-1.23E89	_____
Q	_____
00.100	_____

Did you notice how the screen got messed up when you entered Q? The computer was expecting a numerical value, but it got a string instead. There is a way to prevent input mistakes, like the one above, from ruining a well-planned screen display.

Change statement 20 to:

```
20 ACCEPT AT(2,20)VALIDATE(NUMERIC):M
```

RUN the program again. Try several different inputs to see what happens.

DATA	RESPONSE

The VALIDATE(NUMERIC) option only allows numeric data to be entered (0 through 9,".","+","-", and "E").

Other examples of the VALIDATE option are shown below:

VALIDATE (*ualpha*) — allows only uppercase alphabetic characters.

VALIDATE(*digit*) — allows only 0 through 9.

VALIDATE(*string expression*) — allows only the characters in the given string expression.

Now change statements 20 and 30 to:

```
20 ACCEPT AT(2,20)BEEP VALIDATE("YESNO"):M$
30 DISPLAY AT(20,2)BEEP:M$
```

RUN the program again. Input the following data:

DATA	RESPONSE
HELLO	_____
YESNO	_____
YES	_____
NO	_____
Y	_____
YO	_____
12	_____

What happens if statement 20 is typed in as:

```
20 ACCEPT AT(2,20)VALIDATE("YESNO") BEEP:M$
```

Now let's investigate another option available with the ACCEPT AT command.

Type in the following program and then RUN it.

```
10 CALL CLEAR
20 DISPLAY AT(2,1):"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 FOR I=1 TO 1000::NEXT I
40 ACCEPT AT(2,10):A$
50 DISPLAY AT(20,10):A$
```

Notice that statement 40 erases the rest of the second row beginning at column 10 and then waits for your input.

One can modify this erasure by using the SIZE command.

Change statement 40 to:

```
40 ACCEPT AT(2,10)SIZE(5):A$
```

RUN the program again, noticing the effect of the SIZE option.

Now change statement 40 to:

```
40 ACCEPT AT(2,10)SIZE(-5):A$
```

RUN the program again. When the program is waiting for a response from you, just press ENTER.

This time statement 40 does not clear the 5 spaces. If you input a string, the computer will accept up to 5 characters. If no new string is typed onto the screen, the computer accepts whatever is in those 5 spaces.

Finally, change statement 40 to:

```
40 ACCEPT AT(2,10)BEEP VALIDATE(DIGIT,"YN")SIZE(-1)
ERASE ALL:A$
```

RUN the program again.

ERASE ALL causes the whole screen to be cleared.

EXERCISE 25-1

In this exercise, you are to write a program using the DISPLAY, ACCEPT, and IMAGE commands.

Your program should input numbers from the screen using the ACCEPT AT statement. The program should BEEP every time it is waiting for an input. The input statement should VALIDATE that the data entered is numeric.

As each number is entered, the program should display the current sum of the numbers entered. Also, have it print out the number of data entries.

When you wish to zero the adder, enter a 0. The program should respond by clearing the counter and displaying zeroes in each display area.

A possible screen display is shown below.

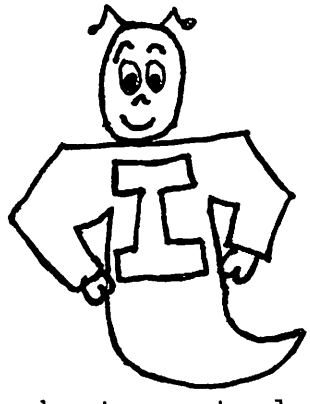
```
THE ADDING MACHING
NUMBER OF ENTRIES: 000
      CURRENT SUM: 000.00

*ENTER 0 TO CLEAR
```

Use an IMAGE statement to produce this number format:

```
XXX.XX
```

Record your working program on the lines provided on the next page.

LESSON #26: SUB



IN THIS LESSON, YOU WILL LEARN HOW TO CREATE YOUR OWN SUBPROGRAMS. SUCH SUBPROGRAMS ARE CALLED JUST LIKE REGULAR TI BASIC OR EXTENDED BASIC SUBPROGRAMS.

The best way to learn about subprogramming is just to crawl right in. So here we go. . .

Type the following program into the computer:

```
10 CALL CLEAR
20 CALL ADDER(1,4,A)
30 CALL ADDER(12,3,B)
40 PRINT A
50 PRINT B
100 SUB ADDER(NUM1,NUM2,SUM)
110 SUM=NUM1+NUM2
120 SUBEND
```

RUN the program.

There are several things you should notice. The program begins by clearing the screen (line 10). Statement 20 then calls a subprogram named ADDER. This is a user defined subprogram. It contains 3 variables, 2 of which are passed to the subprogram, and 1 which is given a value by the subprogram.

After statement 20 is executed, the program jumps down to (passes control to) the subprogram which begins in line 100.

Statements 100 through 120 define the subprogram's operation. Statement 100 tells the computer that a subprogram is being defined, that its name is ADDER, and that it has 3 variables.

When statement 100 is executed, NUM1 and NUM2 are set equal to the numbers passed from the main program. That is, NUM1=1 and NUM2=4.

Next, statement 110 computes the value of SUM. SUM=1+4=5. This value is associated with A (A=SUM=5).

Statement 120 signals the end of the subprogram's operation. Control is passed back to the main program at line 30.

Statement 30 makes another call to the subprogram. This time NUM1=12, NUM2=3, and SUM=15=B.

Control is passed from the subprogram back to line 40.

Statements 40 and 50 print the values of A and B that have been generated.

The program stops after statement 50.

NOTE: SUBPROGRAMS SHOULD ALWAYS COME AT THE END OF THE MAIN PROGRAM.

Now type the following program into the computer and RUN it.

```
10 CALL CLEAR
20 X=10
30 CALL PROG
40 PRINT "MAIN PROGRAM VALUE OF X=";X
50 GOTO 50
60 SUB PROG
70 X=50
80 PRINT "SUBPROGRAM VALUE OF X=";X
90 SUBEND
```

Here's how the program works:

After line 10 clears the screen, line 120 sets X equal to 10.

Statement 30 is executed next. It calls the subprogram named PROG. Notice that no variables are passed between the main program and the subprogram.

Statements 60, 70, 80, and 90 are executed next. Line 70 sets X equal to 50 and line 80 prints this value.

Line 90 signals the end of the subprogram and sends control back to line 40 which prints the value of X.

Notice! Instead of printing a value of 50 for X, line 40 prints the earlier value, 10. The reason for this is that the computer considers the X in the main program and the X in the subprogram to be different variables.

This is a great advantage! It means that you don't have to worry that you've mistakenly used the same variable in a main program as in a subprogram.

It also means that you can use the same subprogram with many different main programs without having to change the variable names within the subprogram.

The variables within a subprogram are said to be local variables since they are only "known" to the subprogram.

Now type the following program into the computer and RUN it.

```
10 CALL CLEAR::A=5
20 CALL PROG(A)
30 PRINT A
40 GOTO 40
50 SUB PROG(X)
60 A=10
70 PRINT X
80 SUBEND
```

Record the program display: line 70 _____

LINE 30 _____

Now change statement 50 to:

```
50 SUB PROG(A)
```

RUN the program again. Record the program display:

line 70 _____

line 30 _____

Finally change statement 20 to:

```
20 CALL PROG ((A))
```

RUN the program again. Record the display as before:

line 7Ø _____

line 30 _____

Finally, make the following program changes:

```
20 CALL PROG(A)
50 SUB PROG (X)
60 X=10
```

LIST the program to observe the complete program, then RUN it. Again, record the display:

line 70 _____

line 30 _____

Notice under what conditions the value of A(=5) in the main program gets altered by the subprogram.

Now input and RUN the program listed below.

```
10 CALL SAY("ENTER X")
20 INPUT X
30 CALL A(X)
40 PRINT "MAIN"
50 GOTO 10
60 SUB A(M)
70 IF M=5 THEN PRINT "SUBEXIT"::SUBEXIT
```

Keep going. ——————→

```
80 PRINT "SUBEND"
90 SUBEND
```

This program shows a way to leave a subprogram before the SUBEND statement.

Input the following values of X and record the computer's response.

X	RESPONSE
1	_____
5	_____

NOTE: THE VOCABULARY FOR THE CALL SAY SUBPROGRAM HAS BEEN GIVEN IN AN EARLIER VOLUME.

EXERCISE 26-1

Write a main program that ACCEPTs 4 numbers from the keyboard and then calls a subprogram which uses these 4 numbers as variables. After the main program calls the subprogram, it should loop back to the ACCEPT statement waiting for more inputs.

The subprogram should interpret the 4 variables as follows:

1st variable — The color of solid blocks to be used in building up a larger square.

2nd variable — The screen row position of the upper left-hand corner of the large square.

3rd variable — The screen column position of the upper left-hand corner of the large square.

4th variable — The length (in blocks) of each side of the square.

The subprogram should draw a square on the screen with the above specifications.

Use IF-THEN statements to cause the subprogram to execute a SUBEXIT if any given specifications are outside the allowed range.

Show your working program on the lines below.

EXERCISE 26-2

Type the following program into the computer and RUN it.

```
10 ON ERROR 100
20 INPUT X
30 CALL SCREEN (X)
40 PRINT X
50 GOTO 20
100 PRINT "ERROR"
110 RETURN 20
```

Input the following data. Record the computer's response:

X	RESPONSE
7	_____
14	_____
17	_____
13	_____
18	_____

Statement 10 tells the computer what to do if an error occurs. In this case, the computer will be directed to a subroutine beginning at statement 100.

After you input a 17 at line 20, an error occurs in line 30. The computer immediately jumps to line 100 to begin execution of the error handling routine. The routine prints "ERROR" and then returns to line 20 of the main program. (If no line number is given after RETURN, it will cause a return to the same line that causes the error.)

Keep going. ——————→

EXERCISE 26-2 (CONT.)

Later, when you input 18 at line 20, an error condition again arises at line 30. This time, however, the program stops with an error message. The reason for this is that the second time an error occurs, the ON ERROR (statement number) is no longer in effect. The computer reverts to its normal action: ON ERROR STOP.

To circumvent this, you need to reset the error option after every error.

Add the following program line and RUN the program again using the data below.

```
105 ON ERROR 100
```

X	RESPONSE
7	_____
14	_____
17	_____
13	_____
18	_____

Now make the following change:

```
110 RETURN NEXT
```

RUN the program again. What is the effect of the change made in line 110? _____

_____

Keep going. ——————→

EXERCISE 26-2 (CONT.)

Now use the ON ERROR (line number) statement in the program that you wrote in EXERCISE 26-1. Use it in place of the IF-THEN statements to keep the program from crashing when error conditions arise. Show your working program on the lines below.

THE COLORED PAGES

At the end of this manual, you will find several colored pages. These are projects that test your ability to use what you have learned. There are no right or wrong answers. If your program does what is asked, then it is quite acceptable. You are free to express your creativity. Be proud of what you do. Do not worry whether your solution is like anyone else's.

Some of these projects may seem easy. . .but do not be deceived into thinking that you can skip them. After all, if they are easy for you, then it will not take long to do them.

Good luck!

Henry A. Taitt

Henry A. Taitt
Director

VIOLET PROJECT 1

Create a program that will cause a sprite to "fly" across the screen. Have the sprite take on all possible character numbers from 32 to 143, with each passage across the screen being a different character number.

VIOLET PROJECT 2

Create a program that causes a sprite to "fall" from the top of the screen to the bottom. Have the velocity increase like a real freely falling object.

$$V = V_o + a * t$$

where: V = Velocity

V_o = Original Velocity

a = acceleration

t = time

You may wish to review the YELLOW ALL STAR book, pages 31 to 33.

VIOLET PROJECT 3

Using DISPLAY and ACCEPT, create a program of your own design that also uses the SUB command.

VIOLET PROJECT 4

Using subprograms, create a program that will take the number that you INPUT, and sum all whole numbers (integers) between it and zero IF you INPUT an even number.

If you INPUT an odd number, have it sum only the odd number (integers) between it and zero.

VIOLET PROJECT 5

Using four subprograms, create four graphic figures. Give each graphic figure a name. Have them appear randomly at different locations on the screen with their name printed beneath them.

VIOLET PROJECT 6

Design a program that uses high-resolution graphics to create a picture of the American Flag. If you can, have sprites fly across the flag saying:

"OH SAY CAN YOU SEE?"

Copy your program on tape or disk and send it along with this page to obtain your Programmer VII card. You have become an excellent programmer! Congratulations!

The cost of creating for you a PROGRAMMER'S card is included in the cost of this manual. By sending in this colored page, we know this cost has already been paid. If you are sharing this manual and cannot remove this page, then you may also receive your PROGRAMMER'S card and have your name listed in the newsletter by sending us a copy of your working program for this project along with $2.00 and a self-addressed, stamped envelope.

Send to:

Henry A. Taitt
CREATIVE Programming, Inc.
604 Sixth Street
Charleston, IL 61920

TI-99/4A

Your name _____

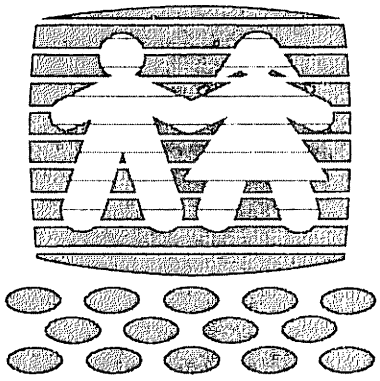
Phone # _____

Address _____

City, State _____

Zip _____ Birthdate _____

Don't forget to enclose a self-addressed stamped envelope.



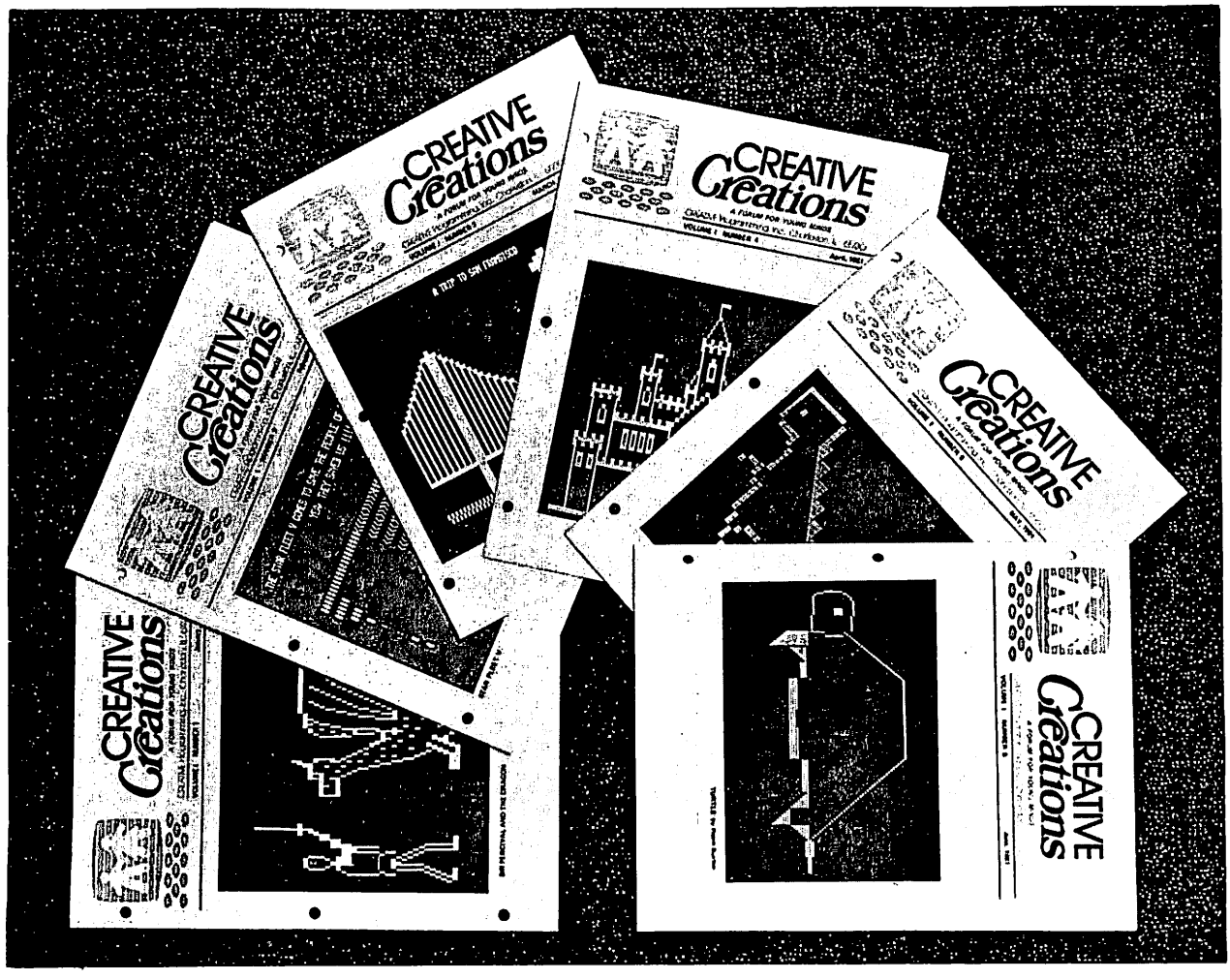
CREATIVE Creations

A FORUM FOR YOUNG MINDS

CREATIVE Programming, Inc., Charleston, IL 61920

A newsletter published 12 times a year. The articles are for young programmers, about young programmers and often written by young programmers.

Each month a graphics program created by a student is selected for the cover. It could be yours! Contests, mind bending challenges, computer game reviews, new creations, programs, even an X-rated column for parents and teachers who are running programs in their areas.



Name_____

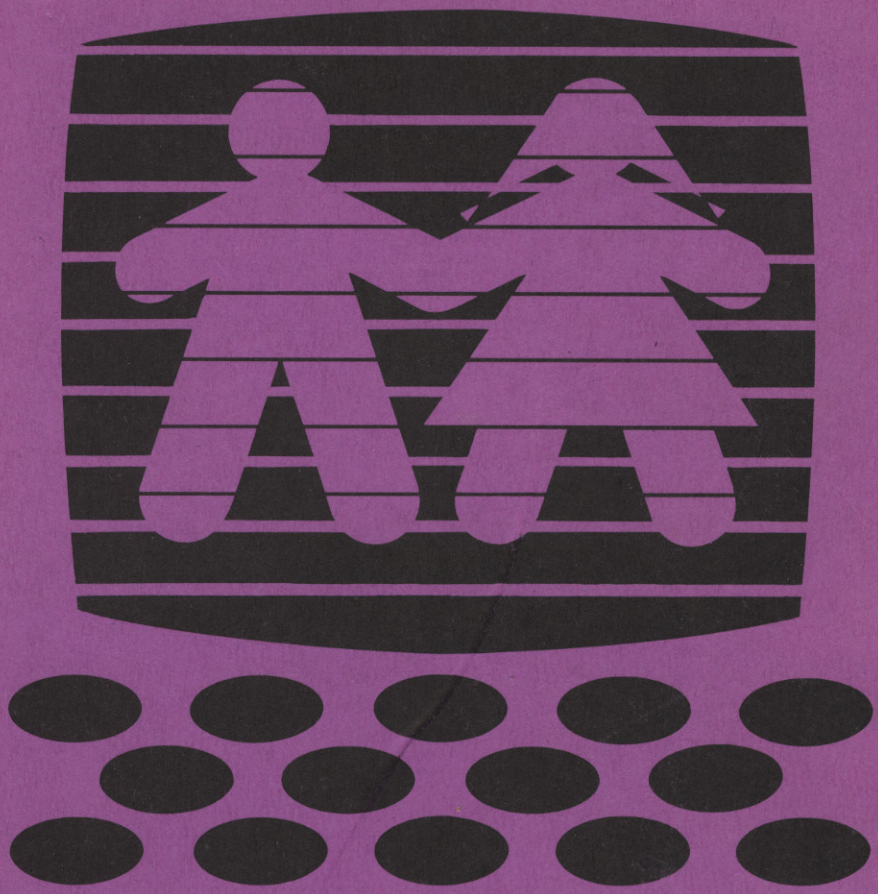
Address _____

City _____ State _____ Zip _____

Please make checks payable to: CREATIVE Creations
604 Sixth Street
Charleston, IL 61920

Only $18 a year ($32 for two years) brings all twelve issues to your door. Join us today in sharing in the excitement of CREATIVE Programming through CREATIVE Creations.

☐ one year ($18.00) ☐ two years ($32.00)



CREATIVE PROGRAMMING INCORPORATED
A SUBSIDIARY OF R.V. WEATHERFORD CO.

604 6th St., Charleston, IL 61920

T19-R-07